Lecture 11: Ensembles

UBC 2022-23

Instructor: Mathias Lécuyer

> The interests of truth require a diversity of opinions.
> by John Stuart Mill

Imports

In [1]:

```
import os

%matplotlib inline
import string
import sys
from collections import deque

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

sys.path.append("../code/.")

from plotting_functions import *
from sklearn import datasets
from sklearn.compose import ColumnTransformer, make_column_transformer
from sklearn.dummy import DummyClassifier, DummyRegressor
from sklearn.ensemble import RandomForestClassifier, RandomForestRegres
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import (
    GridSearchCV,
    RandomizedSearchCV,
    cross_val_score,
    cross_validate,
    train_test_split,
)
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder, Standa
from sklearn.svm import SVC, SVR
from sklearn.tree import DecisionTreeClassifier
from utils import *
```

Lecture learning objectives

From this lecture, you will be able to

- Use `scikit-learn`'s `RandomForestClassifier` and explain its main hyperparameters.
- Explain randomness in the random forest algorithm.
- Use other tree-based models such as as `XGBoost` and `LGBM`.
- Employ ensemble classifier approaches, in particular model averaging and stacking.
- Explain voting and stacking and the differences between them.
- Use `scikit-learn` implementations of these ensemble methods.

Motivation

- **Ensembles** are models that combine multiple machine learning models to create more powerful models.

The Netflix prize

Source (https://netflixtechblog.com/netflix-recommendations-beyond-the-5-stars-part-1-55838468f429)

- Most of the winning solutions for Kaggle competitions involve some kind of ensembling. For example:

Key idea: Groups can often make better decisions than individuals, especially when group members are diverse enough.

The Wisdom of Crowds (http://wisdomofcrowds.blogspot.com/2009/12/introduction-part-i.html)

Tree-based ensemble models

- A number of ensemble models in ML literature.
- Most successful ones on a variety of datasets are tree-based models.
- We'll briefly talk about two such models:
  - Random forests
  - Gradient boosted trees
- We'll also talk about averaging and stacking.

Tree-based models

- Decision tree models are
  - Interpretable
  - They can capture non-linear relationships
  - They don't require scaling of the data and theoretically can work with categorical features.
- But a single decision tree is likely to overfit.

- Key idea: Combine multiple trees to build stronger models.
- These kinds of models are extremely popular in industry and machine learning competitions

Data

- Let's work with the adult census data set (https://www.kaggle.com/uciml/adult-census-income).

In [2]:
```
adult_df_large = pd.read_csv("../data/adult.csv")
train_df, test_df = train_test_split(adult_df_large, test_size=0.2, ran
train_df_nan = train_df.replace("?", np.NaN)
test_df_nan = test_df.replace("?", np.NaN)
train_df_nan.head()
```

Out[2]:

	age	workclass	fnlwgt	education	education.num	marital.status	occupation	relationship	
5514	26	Private	256263	HS-grad	9	Never-married	Craft-repair	Not-in-family	\
19777	24	Private	170277	HS-grad	9	Never-married	Other-service	Not-in-family	\
10781	36	Private	75826	Bachelors	13	Divorced	Adm-clerical	Unmarried	\
32240	22	State-gov	24395	Some-college	10	Married-civ-spouse	Adm-clerical	Wife	\
9876	31	Local-gov	356689	Bachelors	13	Married-civ-spouse	Prof-specialty	Husband	\

In [3]:
```
numeric_features = ["age", "fnlwgt", "capital.gain", "capital.loss", "h
categorical_features = [
    "workclass",
    "marital.status",
    "occupation",
    "relationship",
    "native.country",
]
ordinal_features = ["education"]
binary_features = ["sex"]
drop_features = ["race", "education.num"]
target_column = "income"
```

In [4]:
```
education_levels = [
    "Preschool",
    "1st-4th",
    "5th-6th",
    "7th-8th",
    "9th",
    "10th",
    "11th",
    "12th",
    "HS-grad",
    "Prof-school",
    "Assoc-voc",
    "Assoc-acdm",
    "Some-college",
    "Bachelors",
    "Masters",
    "Doctorate",
]
```

In [5]:
```
assert set(education_levels) == set(train_df["education"].unique())
```

In [6]:
```
numeric_transformer = make_pipeline(StandardScaler())

ordinal_transformer = make_pipeline(
    OrdinalEncoder(categories=[education_levels], dtype=int)
)

categorical_transformer = make_pipeline(
    SimpleImputer(strategy="constant", fill_value="missing"),
    OneHotEncoder(handle_unknown="ignore", sparse=False),
)

binary_transformer = make_pipeline(
    SimpleImputer(strategy="constant", fill_value="missing"),
    OneHotEncoder(drop="if_binary", dtype=int),
)

preprocessor = make_column_transformer(
    (numeric_transformer, numeric_features),
    (ordinal_transformer, ordinal_features),
    (binary_transformer, binary_features),
    (categorical_transformer, categorical_features),
    ("drop", drop_features),
)
```

In [7]:
```
X_train = train_df_nan.drop(columns=[target_column])
y_train = train_df_nan[target_column]

X_test = test_df_nan.drop(columns=[target_column])
y_test = test_df_nan[target_column]
```

Do we have class imbalance?

- There is class imbalance. But without any context, both classes seem equally important.
- Let's use accuracy as our metric.

In [8]:
```
train_df_nan["income"].value_counts(normalize=True)
```

Out[8]:
```
<=50K    0.757985
>50K     0.242015
Name: income, dtype: float64
```

In [9]:
```
scoring_metric = "accuracy"
```

Let's store all the results in a dictionary called `results` .

In [10]:
```
results = {}
```

Baselines

`DummyClassifier` baseline

In [11]:
```
dummy = DummyClassifier(strategy="stratified")
results["Dummy"] = mean_std_cross_val_scores(
    dummy, X_train, y_train, return_train_score=True, scoring=scoring_m
)
```

`DecisionTreeClassifier` baseline

- Let's try decision tree classifier on our data.

In [12]:
```
pipe_dt = make_pipeline(preprocessor, DecisionTreeClassifier(random_sta
results["Decision tree"] = mean_std_cross_val_scores(
    pipe_dt, X_train, y_train, return_train_score=True, scoring=scoring
)
pd.DataFrame(results).T
```

Out[12]:

	fit_time	score_time	test_score	train_score
Dummy	0.008 (+/- 0.001)	0.005 (+/- 0.000)	0.632 (+/- 0.004)	0.633 (+/- 0.003)
Decision tree	0.152 (+/- 0.006)	0.015 (+/- 0.001)	0.813 (+/- 0.003)	1.000 (+/- 0.000)

Decision tree is clearly overfitting.

Random forests

General idea

- A single decision tree is likely to overfit
- Random forests use a collection of diverse decision trees
- Each tree overfits on some part of the data but we can reduce overfitting by averaging the results
  - can be shown mathematically

`RandomForestClassifier`

- Before understanding the details let's first try it out.

In [13]:
```
from sklearn.ensemble import RandomForestClassifier

pipe_rf = make_pipeline(
    preprocessor, RandomForestClassifier(random_state=123, n_jobs=-1)
)
results["Random forests"] = mean_std_cross_val_scores(
    pipe_rf, X_train, y_train, return_train_score=True, scoring=scoring
)
pd.DataFrame(results).T
```

Out[13]:

	fit_time	score_time	test_score	train_score
Dummy	0.008 (+/- 0.001)	0.005 (+/- 0.000)	0.632 (+/- 0.004)	0.633 (+/- 0.003)
Decision tree	0.152 (+/- 0.006)	0.015 (+/- 0.001)	0.813 (+/- 0.003)	1.000 (+/- 0.000)
Random forests	1.474 (+/- 2.339)	0.041 (+/- 0.004)	0.857 (+/- 0.004)	1.000 (+/- 0.000)

The validation scores are better although it seems likes we are still overfitting.

How do they work?

- Decide how many decision trees we want to build
  - can control with `n_estimators` hyperparameter
- `fit` a diverse set of that many decision trees by **injecting randomness** in the classifier construction
- `predict` by voting (classification) or averaging (regression) of predictions given by individual models

Inject randomness in the classifier construction

To ensure that the trees in the random forest are different we inject randomness in two ways:

1. Data: **Build each tree on a bootstrap sample** (i.e., a sample drawn **with replacement** from the training set)
2. Features: **At each node, select a random subset of features** (controlled by `max_features` in `scikit-learn`) and look for the best possible split involving one of these features

An example of a bootstrap samples Suppose this is your original dataset: [1,2,3,4]

- a sample drawn with replacement: [1,1,3,4]
- a sample drawn with replacement: [3,2,2,2]
- a sample drawn with replacement: [1,2,4,4]
- ...

There is also something called `ExtraTreesClassifier` (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html), where we add more randomness by considering a random subset of features at each split and a the **best threshold among a set of random thresholds**.

The random forests classifier

- Create a collection (ensemble) of trees. Grow each tree on an independent bootstrap sample from the data.
- At each node:
  - Randomly select a subset of features out of all features (independently for each node).
  - Find the best split on the selected features.
  - Grow the trees to maximum depth.
- At prediction time:
  - Vote the trees to get predictions for new example.

Example

- Let's create a random forest with 3 estimators.
- Using `max_depth=2` for easy visualization.

In [14]:
```
pipe_rf_demo = make_pipeline(
    preprocessor, RandomForestClassifier(max_depth=2, n_estimators=3, r
)
pipe_rf_demo.fit(X_train, y_train);
```

- Let's get the feature names of transformed features.

In [15]:
```
feature_names = (
    numeric_features
    + ordinal_features
    + binary_features
    + list(
        pipe_rf_demo.named_steps["columntransformer"]
        .named_transformers_["pipeline-4"]
        .named_steps["onehotencoder"]
        .get_feature_names_out()
    )
)
feature_names[:10]
```

Out[15]:
```
['age',
 'fnlwgt',
 'capital.gain',
 'capital.loss',
 'hours.per.week',
 'education',
 'sex',
 'x0_Federal-gov',
 'x0_Local-gov',
 'x0_Never-worked']
```

- Let's sample a test example.

In [16]:
```
test_example = X_test.sample(1)
print("Classes: ", pipe_rf_demo.classes_)
print("Prediction by random forest: ", pipe_rf_demo.predict(test_exampl
transformed_example = preprocessor.transform(test_example)
pd.DataFrame(data=transformed_example.flatten(), index=feature_names)
```

```
Classes:  ['<=50K' '>50K']
Prediction by random forest:  ['<=50K']
```

Out[16]:

	0
age	-0.333171
fnlwgt	-0.924948
capital.gain	0.499907
capital.loss	-0.217680
hours.per.week	-0.042081
...	...
x4_Trinadad&Tobago	0.000000
x4_United-States	1.000000
x4_Vietnam	0.000000
x4_Yugoslavia	0.000000
x4_missing	0.000000

86 rows × 1 columns

- We can look at different trees created by the random forest.
- Note that each tree looks at different set of features and slightly different datasets.

In [55]:
```
for i, tree in enumerate(
    pipe_rf_demo.named_steps["randomforestclassifier"].estimators_
):
    print("\n\nTree", i + 1)
    display(display_tree(feature_names, tree))
    print("prediction", tree.predict(preprocessor.transform(test_exampl
```

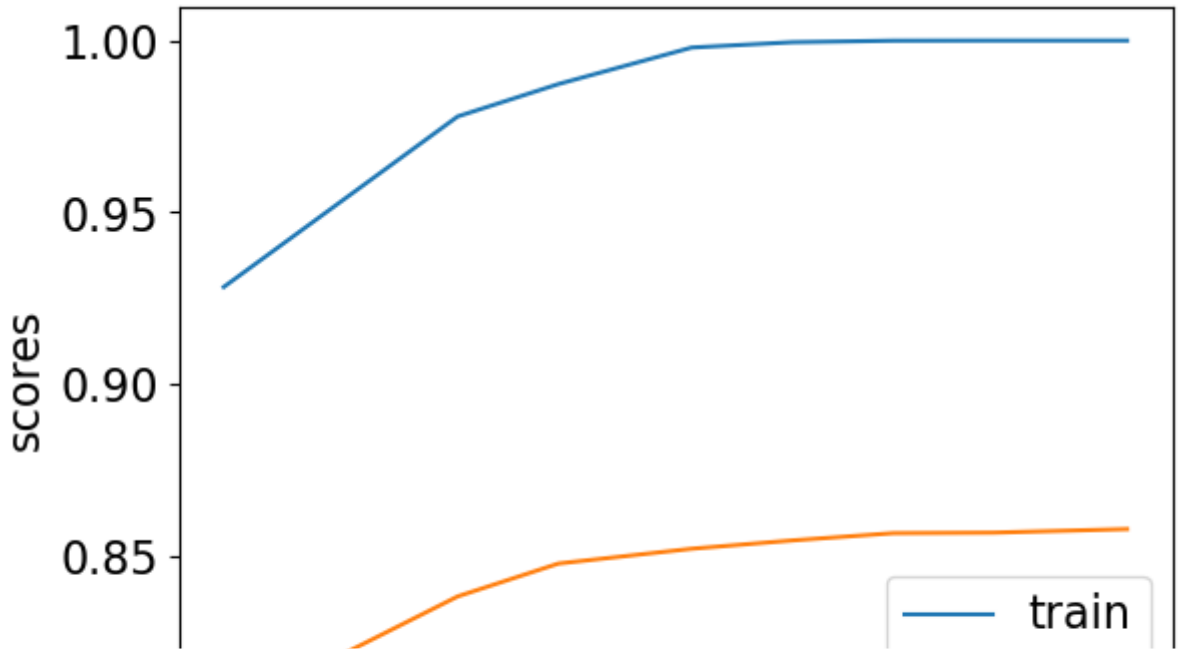
```
Tree 1

<graphviz.sources.Source at 0x199c5bb80>

prediction [0.]


Tree 2

<graphviz.sources.Source at 0x199c5b5b0>

prediction [0.]


Tree 3

<graphviz.sources.Source at 0x199efc730>

prediction [0.]
```

Some important hyperparameters:

- `n_estimators` : number of decision trees (higher = more complexity)
- `max_depth` : max depth of each decision tree (higher = more complexity)
- `max_features` : the number of features you get to look at each split (higher = more complexity; lower = more randomness = more diversity across estimators)

Random forests: number of trees (`n_estimators`) and the fundamental tradeoff

In [56]:
```
make_num_tree_plot(
    preprocessor, X_train, y_train, X_test, y_test, [1, 5, 10, 25, 50,
)
```



Number of trees and fundamental trade-off

- Above: seems like we're beating the fundamental "tradeoff" by increasing training score and not decreasing validation score much.
- This is the promise of ensembles, though it's not guaranteed to work so nicely.

Here, more trees are always better! We pick less trees for speed.

Strengths and weaknesses

- Usually one of the best performing off-the-shelf classifiers without heavy tuning of hyperparameters,
- Doesn't require scaling of data,
- Less likely to overfit,
- Slower than decision trees because we are fitting multiple trees but can easily parallelize training because all trees are independent of each other,
- In general, able to capture a much broader picture of the data compared to a single decision tree.

Weaknesses

- Require more memory,

- Hard to interpret,
- Tend not to perform well on high dimensional sparse data such as text data.

Make sure to set the `random_state` for reproducibility. Changing the `random_state` can have a big impact on the model and the results due to the random nature of these models. Having more trees can get you a more robust estimate.

The original random forests paper (https://www.stat.berkeley.edu/~breiman/randomforest2001.pdf) by Leo Breiman.

Gradient boosted trees

Another popular and effective class of tree-based ensemble models is gradient boosted trees.

- No randomization.
- The key idea is to combine many simple models called weak learners to create a strong learner.
- They combine multiple shallow (depth 1 to 5) decision trees
- They build trees in a serial manner, where each tree tries to correct the mistakes of previous ones.

Important hyperparameters

- `n_estimators`
  - control the number of trees to build
- `learning_rate`
  - controls how strongly each tree tries to correct the mistakes of the previous trees
  - higher learning rate means each tree can make stronger corrections, which means more complex model

We'll not go into the details. We'll look at brief examples of using the following three gradient boosted tree models.

- XGBoost (https://xgboost.readthedocs.io/en/latest/)
- LightGBM (https://lightgbm.readthedocs.io/en/latest/Python-Intro.html)
- CatBoost (https://catboost.ai/docs/concepts/python-quickstart.html)

XGBoost (https://xgboost.ai/about)

- Not part of `sklearn` but has similar interface.
- Install it in your conda environment: `conda install -c conda-forge xgboost`
- Supports missing values
- GPU training, networked parallel training
- Supports sparse data
- Typically better scores than random forests

LightGBM (https://lightgbm.readthedocs.io/)

- Not part of `sklearn` but has similar interface.
- Install it in your conda environment: `conda install -c conda-forge lightgbm`
- Small model size
- Faster
- Typically better scores than random forests

CatBoost (https://catboost.ai/)

- Not part of `sklearn` but has similar interface.
- Install it in your conda environment: `conda install -c conda-forge catboost`
- Usually better scores but slower compared to `XGBoost` and `LightGBM`

In [57]:
```
from catboost import CatBoostClassifier
from lightgbm.sklearn import LGBMClassifier
from sklearn.tree import DecisionTreeClassifier
from xgboost import XGBClassifier

from sklearn.compose import TransformedTargetRegressor
from sklearn.preprocessing import LabelEncoder

pipe_lr = make_pipeline(
    preprocessor, LogisticRegression(max_iter=2000, random_state=123)
)
pipe_dt = make_pipeline(preprocessor, DecisionTreeClassifier(random_sta
pipe_rf = make_pipeline(preprocessor, RandomForestClassifier(random_sta
pipe_xgb = make_pipeline(
    preprocessor, XGBClassifier(random_state=123, eval_metric="logloss"
)
pipe_lgbm = make_pipeline(preprocessor, LGBMClassifier(random_state=123
pipe_catboost = make_pipeline(
    preprocessor, CatBoostClassifier(verbose=0, random_state=123)
)
# XGBoost requires numeric targets
label_encoder = LabelEncoder()
label_encoder.fit(y_train)

classifiers = {
    "logistic regression": pipe_lr,
    "decision tree": pipe_dt,
    "random forest": pipe_rf,
    "XGBoost": pipe_xgb,
    "LightGBM": pipe_lgbm,
    "CatBoost": pipe_catboost,
}
```

In [58]:
```
import warnings

warnings.simplefilter(action="ignore", category=DeprecationWarning)
warnings.simplefilter(action="ignore", category=UserWarning)
```

In [59]:
```
results = {}
```

In [60]:
```
dummy = DummyClassifier(strategy="stratified")
results["Dummy"] = mean_std_cross_val_scores(
    dummy, X_train, y_train, return_train_score=True, scoring=scoring_m
)
```

In [61]:
```
for (name, model) in classifiers.items():
    results[name] = mean_std_cross_val_scores(
        model, X_train, label_encoder.transform(y_train), return_train_
    )
```

In [62]:
```
pd.DataFrame(results).T
```

Out[62]:

	fit_time	score_time	test_score	train_score
Dummy	0.007 (+/- 0.000)	0.006 (+/- 0.000)	0.631 (+/- 0.006)	0.634 (+/- 0.003)
logistic regression	1.316 (+/- 0.044)	0.010 (+/- 0.000)	0.850 (+/- 0.006)	0.851 (+/- 0.001)
decision tree	0.144 (+/- 0.002)	0.010 (+/- 0.001)	0.813 (+/- 0.003)	1.000 (+/- 0.000)
random forest	1.398 (+/- 0.028)	0.082 (+/- 0.002)	0.857 (+/- 0.004)	1.000 (+/- 0.000)
XGBoost	1.066 (+/- 0.031)	0.015 (+/- 0.000)	0.870 (+/- 0.003)	0.909 (+/- 0.002)
LightGBM	0.740 (+/- 0.062)	0.016 (+/- 0.001)	0.871 (+/- 0.004)	0.892 (+/- 0.000)
CatBoost	4.566 (+/- 0.075)	0.079 (+/- 0.001)	0.872 (+/- 0.003)	0.900 (+/- 0.001)

Some observations

- Keep in mind all these results are with default hyperparameters
- Ideally we would carry out hyperparameter optimization for all of them and then compare the results.
- We are using a particular scoring metric (accuracy in this case)
- We are scaling numeric features but it shouldn't matter for these tree-based models.
- Look at the std. Doesn't look very high.
    - The scores look more or less stable.

In [63]:
```
pd.DataFrame(results).T
```

Out[63]:

	fit_time	score_time	test_score	train_score
Dummy	0.007 (+/- 0.000)	0.006 (+/- 0.000)	0.631 (+/- 0.006)	0.634 (+/- 0.003)
logistic regression	1.316 (+/- 0.044)	0.010 (+/- 0.000)	0.850 (+/- 0.006)	0.851 (+/- 0.001)
decision tree	0.144 (+/- 0.002)	0.010 (+/- 0.001)	0.813 (+/- 0.003)	1.000 (+/- 0.000)
random forest	1.398 (+/- 0.028)	0.082 (+/- 0.002)	0.857 (+/- 0.004)	1.000 (+/- 0.000)
XGBoost	1.066 (+/- 0.031)	0.015 (+/- 0.000)	0.870 (+/- 0.003)	0.909 (+/- 0.002)
LightGBM	0.740 (+/- 0.062)	0.016 (+/- 0.001)	0.871 (+/- 0.004)	0.892 (+/- 0.000)
CatBoost	4.566 (+/- 0.075)	0.079 (+/- 0.001)	0.872 (+/- 0.003)	0.900 (+/- 0.001)

- Decision trees and random forests overfit
    - Other models do not seem to overfit much.
- Fit times
    - Decision trees are fast but not very accurate
    - LightGBM is faster than decision trees and more accurate!
    - CatBoost fit time is highest followed by random forests.
    - There is not much difference between the validation scores of XGBoost, LightGBM, and CatBoost but it is about 48x slower than LightGBM!
    - XGBoost and LightGBM are faster and more accurate than random forest!
- Scores times

- Prediction times are much smaller in all cases.

What classifier should I use?

Simple answer

- Whichever gets the highest CV score making sure that you're not overusing the validation set.

Interpretability

- This is an area of growing interest and concern in ML.
- How important is interpretability for you?
- In the next class we'll talk about interpretability of non-linear models.

Speed/code maintenance

- Other considerations could be speed (fit and/or predict), maintainability of the code.

Finally, you could use all of them!

Averaging

Earlier we looked at a bunch of classifiers:

In [64]:
```
classifiers.keys()
```

Out[64]:
```
dict_keys(['logistic regression', 'decision tree', 'random forest', 'XGBo
ost', 'LightGBM', 'CatBoost'])
```

What if we use all these models and let them vote during prediction time?

In [65]:
```
from sklearn.ensemble import VotingClassifier

averaging_model = VotingClassifier(
    list(classifiers.items()), voting="soft"
)  # need the list() here for cross_val to work!
```

In [66]:

```
from sklearn import set_config

set_config(display="diagram")  # global setting
```

In [67]:
```
1 averaging_model
```

Out[67]:
```
VotingClassifier(estimators=[('logistic regression',
                              Pipeline(steps=[('columntransformer',
                                               ColumnTransformer(transfor
mers=[('pipeline-1',

Pipeline(steps=[('standardscaler',

StandardScaler())]),

['age',

'fnlwgt',

'capital.gain',

'capital.loss',

'hours.per.week']),

('pipeline-2',

Pipeline(steps=[('ordinalencoder',

OrdinalEncoder(categories=[['Preschool',

'1st-4th'...

Pipeline(steps=[('simpleimputer',

SimpleImputer(fill_value='missing',

strategy='constant')),

('onehotencoder',

OneHotEncoder(handle_unknown='ignore',

sparse=False))]),

['workclass',

'marital.status',

'occupation',

'relationship',

'native.country']),
```

```
('drop',

'drop',

['race',

'education.num'])])),
                                                    ('catboostclassifier',
                                                     <catboost.core.CatBoostCla
ssifier object at 0x19aca82b0>)]))],
                  voting='soft')
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

This `VotingClassifier` will take a *vote* using the predictions of the constituent classifier pipelines.

Main parameter: `voting`

- `voting='hard'`
  - it uses the output of `predict` and actually votes.
- `voting='soft'`
  - with `voting='soft'` it averages the output of `predict_proba` and then thresholds / takes the larger.

- The choice depends on whether you trust `predict_proba` from your base classifiers - if so, it's nice to access that information.

In [68]:
```
1 averaging_model.fit(X_train, y_train);
```

- What happens when you `fit` a `VotingClassifier`?
  - It will fit all constituent models.

```
It seems sklearn requires us to actually call `fit` on the `VotingC
lassifier`, instead of passing in pre-fit models. This is an implem
entation choice rather than a conceptual limitation.
```

Let's look at particular test examples where `income` is ">50k" (y=1):

In [69]:
```
test_g50k = (
    test_df.query("income == '>50K'").sample(4, random_state=2).drop(co
)
test_l50k = (
    test_df.query("income == '<=50K'")
    .sample(4, random_state=2)
    .drop(columns=["income"])
)
```

In [70]:
```
averaging_model.classes_
```

Out[70]:
```
array(['<=50K', '>50K'], dtype=object)
```

In [71]:
```
data = {"Voting classifier": averaging_model.predict(test_g50k)}
pd.DataFrame(data)
```

Out[71]:

	Voting classifier
0	>50K
1	>50K
2	>50K
3	<=50K

For hard voting, these are the votes:

In [72]:
```
r1 = {
    name: classifier.predict(test_g50k)
    for name, classifier in averaging_model.named_estimators_.items()
}
data.update(r1)
pd.DataFrame(data)
```

Out[72]:

	Voting classifier	logistic regression	decision tree	random forest	XGBoost	LightGBM	CatBoost
0	>50K	1	1	1	1	1	1
1	>50K	1	1	1	1	1	1
2	>50K	1	0	1	1	1	1
3	<=50K	0	0	0	0	0	0

For soft voting, these are the scores:

In [73]:
```
r1 = {
    name: classifier.predict_proba(test_g50k)
    for name, classifier in averaging_model.named_estimators_.items()
}
r1
```

Out[73]:
```
{'logistic regression': array([[2.28705943e-14, 1.00000000e+00],
       [4.18744244e-01, 5.81255756e-01],
       [4.96637268e-01, 5.03362732e-01],
       [8.87444717e-01, 1.12555283e-01]]),
 'decision tree': array([[0., 1.],
       [0., 1.],
       [1., 0.],
       [1., 0.]]),
 'random forest': array([[0.  , 1.  ],
       [0.31, 0.69],
       [0.37, 0.63],
       [0.57, 0.43]]),
 'XGBoost': array([[0.00168145, 0.99831855],
       [0.30067134, 0.69932866],
       [0.31893963, 0.6810604 ],
       [0.78900903, 0.21099098]], dtype=float32),
 'LightGBM': array([[0.00187645, 0.99812355],
       [0.28722892, 0.71277108],
       [0.28457261, 0.71542739],
       [0.8095596 , 0.1904404 ]]),
 'CatBoost': array([[0.00136353, 0.99863647],
       [0.26605365, 0.73394635],
       [0.32779854, 0.67220146],
       [0.82750574, 0.17249426]])}
```

(Aside: the probability scores from `DecisionTreeClassifier` are pretty bad)

Let's see how well this model performs.

In [74]:
```
results["Voting"] = mean_std_cross_val_scores(averaging_model, X_train,
```

In [75]:
```
pd.DataFrame(results).T
```

Out[75]:

	fit_time	score_time	test_score	train_score
Dummy	0.007 (+/- 0.000)	0.006 (+/- 0.000)	0.631 (+/- 0.006)	0.634 (+/- 0.003)
logistic regression	1.316 (+/- 0.044)	0.010 (+/- 0.000)	0.850 (+/- 0.006)	0.851 (+/- 0.001)
decision tree	0.144 (+/- 0.002)	0.010 (+/- 0.001)	0.813 (+/- 0.003)	1.000 (+/- 0.000)
random forest	1.398 (+/- 0.028)	0.082 (+/- 0.002)	0.857 (+/- 0.004)	1.000 (+/- 0.000)
XGBoost	1.066 (+/- 0.031)	0.015 (+/- 0.000)	0.870 (+/- 0.003)	0.909 (+/- 0.002)
LightGBM	0.740 (+/- 0.062)	0.016 (+/- 0.001)	0.871 (+/- 0.004)	0.892 (+/- 0.000)
CatBoost	4.566 (+/- 0.075)	0.079 (+/- 0.001)	0.872 (+/- 0.003)	0.900 (+/- 0.001)
Voting	9.208 (+/- 0.147)	0.222 (+/- 0.003)	0.868 (+/- 0.003)	NaN

It appears that here we didn't do better than our best classifier :(.

Let's try removing decision tree classifier.

In [76]:
```
classifiers_ndt = classifiers.copy()
del classifiers_ndt["decision tree"]
averaging_model_ndt = VotingClassifier(
    list(classifiers_ndt.items()), voting="soft"
)  # need the list() here for cross_val to work!

results["Voting_ndt"] = mean_std_cross_val_scores(
    averaging_model_ndt,
    X_train,
    y_train,
    return_train_score=True,
    scoring=scoring_metric,
)
```

In [77]:
```
pd.DataFrame(results).T
```

Out[77]:

	fit_time	score_time	test_score	train_score
Dummy	0.007 (+/- 0.000)	0.006 (+/- 0.000)	0.631 (+/- 0.006)	0.634 (+/- 0.003)
logistic regression	1.316 (+/- 0.044)	0.010 (+/- 0.000)	0.850 (+/- 0.006)	0.851 (+/- 0.001)
decision tree	0.144 (+/- 0.002)	0.010 (+/- 0.001)	0.813 (+/- 0.003)	1.000 (+/- 0.000)
random forest	1.398 (+/- 0.028)	0.082 (+/- 0.002)	0.857 (+/- 0.004)	1.000 (+/- 0.000)
XGBoost	1.066 (+/- 0.031)	0.015 (+/- 0.000)	0.870 (+/- 0.003)	0.909 (+/- 0.002)
LightGBM	0.740 (+/- 0.062)	0.016 (+/- 0.001)	0.871 (+/- 0.004)	0.892 (+/- 0.000)
CatBoost	4.566 (+/- 0.075)	0.079 (+/- 0.001)	0.872 (+/- 0.003)	0.900 (+/- 0.001)
Voting	9.208 (+/- 0.147)	0.222 (+/- 0.003)	0.868 (+/- 0.003)	NaN
Voting_ndt	9.077 (+/- 0.178)	0.206 (+/- 0.003)	0.872 (+/- 0.004)	0.921 (+/- 0.001)

Still the results are not better than the best performing model.

- It didn't happen here but how could the average do better than the best model???
  - From the perspective of the best estimator (in this case CatBoost), why are you adding on worse estimators??

Here's how this can work:

Example	log reg	rand forest	cat boost	Averaged model
1	✅	✅	❌	✅✅❌ =>✅
2	✅	❌	✅	✅❌✅ =>✅

Example	log reg	rand forest	cat boost	Averaged model

In short, as long as the different models make different mistakes, voting can improve the final predictions.

Why not always do this?

1. `fit` / `predict` time.
2. Reduction in interpretability.
3. Reduction in code maintainability (e.g. Netflix prize).

What kind of estimators can we combine?

- You can combine:
  - completely different estimators, or similar estimators.
  - estimators trained on different samples.
  - estimators with different hyperparameter values.

Stacking

- Another type of ensemble is stacking.
- Instead of averaging the outputs of each estimator, instead use their outputs as *inputs to another model*.
- By default for classification, it uses logistic regression.
  - We don't need a complex model here necessarily, more of a weighted average.
  - The features going into the logistic regression are the classifier outputs, *not* the original features!
  - So the number of coefficients = the number of base estimators!

In [78]:
```
from sklearn.ensemble import StackingClassifier
```

The code starts to get too slow here; so we'll remove CatBoost.

In [79]:
```
classifiers_nocat = classifiers.copy()
del classifiers_nocat["CatBoost"]
```

In [80]:
```
stacking_model = StackingClassifier(list(classifiers_nocat.items()))
```

In [81]:
```
stacking_model.fit(X_train, y_train);
```

What's going on in here?

- It is doing cross-validation by itself by default (see documentation (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.StackingClassifier.html))
  - It is fitting the base estimators on the training fold
  - And the predicting on the validation fold
  - And then fitting the meta-estimator on that output (on the validation fold)

> Note that `estimators_` are fitted on the full X while `final_estimator_` is trained using cross-validated predictions of the base estimators using `cross_val_predict`.

Here is the input features (X) to the meta-model:

In [82]:
```
valid_sample = train_df.sample(4, random_state=2).drop(columns=["income
```

In [83]:
```
r3 = {
    name: pipe.predict_proba(valid_sample)
    for (name, pipe) in stacking_model.named_estimators_.items()
}
r3
```

Out[83]:
```
{'logistic regression': array([[4.33522670e-01, 5.66477330e-01],
       [9.99016977e-01, 9.83023032e-04],
       [8.60803004e-01, 1.39196996e-01],
       [9.95299562e-01, 4.70043768e-03]]),
 'decision tree': array([[1., 0.],
       [1., 0.],
       [1., 0.],
       [1., 0.]]),
 'random forest': array([[0.88, 0.12],
       [1.  , 0.  ],
       [0.95, 0.05],
       [1.  , 0.  ]]),
 'XGBoost': array([[0.75072813, 0.24927188],
       [0.99450225, 0.00549777],
       [0.95364463, 0.04635534],
       [0.9975675 , 0.00243254]], dtype=float32),
 'LightGBM': array([[0.56629894, 0.43370106],
       [0.99263965, 0.00736035],
       [0.91995188, 0.08004812],
       [0.99629847, 0.00370153]])}
```

- Our meta-model is logistic regression (which it is by default).

- Let's look at the learned coefficients.

In [84]:
```
pd.DataFrame(
    data=stacking_model.final_estimator_.coef_[0],
    index=classifiers_nocat.keys(),
    columns=["Coefficient"],
)
```

Out[84]:

	Coefficient
logistic regression	0.763242
decision tree	-0.011344
random forest	0.218986
XGBoost	2.022841
LightGBM	3.684328

In [85]:
```
stacking_model.final_estimator_.intercept_
```

Out[85]:
```
array([-3.31969134])
```

- It seems that the LightGBM is being trusted the most.

In [86]:
```
stacking_model.predict(test_g50k)
```

Out[86]:
```
array(['>50K', '>50K', '>50K', '<=50K'], dtype=object)
```

In [87]:
```
stacking_model.predict_proba(test_g50k)
```

Out[87]:
```
array([[0.03395918, 0.96604082],
       [0.21342258, 0.78657742],
       [0.22864056, 0.77135944],
       [0.88196532, 0.11803468]])
```

(This is the `predict_proba` from logistic regression)

Let's see how well this model performs.

In [88]:
```
results["Stacking_nocat"] = mean_std_cross_val_scores(
    stacking_model, X_train, y_train, return_train_score=True, scoring=
)
```

In [89]:
```
pd.DataFrame(results).T
```

Out[89]:

	fit_time	score_time	test_score	train_score
Dummy	0.007 (+/- 0.000)	0.006 (+/- 0.000)	0.631 (+/- 0.006)	0.634 (+/- 0.003)
logistic regression	1.316 (+/- 0.044)	0.010 (+/- 0.000)	0.850 (+/- 0.006)	0.851 (+/- 0.001)
decision tree	0.144 (+/- 0.002)	0.010 (+/- 0.001)	0.813 (+/- 0.003)	1.000 (+/- 0.000)
random forest	1.398 (+/- 0.028)	0.082 (+/- 0.002)	0.857 (+/- 0.004)	1.000 (+/- 0.000)
XGBoost	1.066 (+/- 0.031)	0.015 (+/- 0.000)	0.870 (+/- 0.003)	0.909 (+/- 0.002)
LightGBM	0.740 (+/- 0.062)	0.016 (+/- 0.001)	0.871 (+/- 0.004)	0.892 (+/- 0.000)
CatBoost	4.566 (+/- 0.075)	0.079 (+/- 0.001)	0.872 (+/- 0.003)	0.900 (+/- 0.001)
Voting	9.208 (+/- 0.147)	0.222 (+/- 0.003)	0.868 (+/- 0.003)	NaN
Voting_ndt	9.077 (+/- 0.178)	0.206 (+/- 0.003)	0.872 (+/- 0.004)	0.921 (+/- 0.001)
Stacking_nocat	24.434 (+/- 0.534)	0.137 (+/- 0.002)	0.872 (+/- 0.004)	0.900 (+/- 0.007)

- The situation here is a bit mind-boggling.
- On each fold of cross-validation it is doing cross-validation.
- This is really loops within loops within loops within loops...

- We can also try a different final estimator:
- Let's `DecisionTreeClassifier` as a final estimator.

In [90]:
```
stacking_model_tree = StackingClassifier(
    list(classifiers_nocat.items()), final_estimator=DecisionTreeClassi
)
```

The results are not very good. But we can look at the tree:

In [91]:
```
stacking_model_tree.fit(X_train, y_train);
```

In [92]:
```
display_tree(list(classifiers_nocat.keys()), stacking_model_tree.final_
```

Out[92]:
```
<graphviz.sources.Source at 0x199f7bc10>
```

An effective strategy

- Randomly generate a bunch of models with different hyperparameter configurations, and then stack all the models.

- What is an advantage of ensembling multiple models as opposed to just choosing one of them?
  - You may get a better score.

- What is an disadvantage of ensembling multiple models as opposed to just choosing one of them?
  - Slower, more code maintenance issues

Summary

- You have a number of models in your toolbox now.
- Ensembles are usually pretty effective.
  - Tree-based classifiers are particularly popular and effective on a wide range of problems.
  - But they trade off code complexity and speed for prediction accuracy.
  - Don't forget that hyperparameter optimization multiplies the slowness of the code!
- Stacking is a bit slower than voting, but generally higher accuracy.
  - As a bonus, you get to see the coefficients for each base classifier.
- All the above models have equivalent regression models.

Relevant papers

- Fernandez-Delgado et al. 2014 (http://jmlr.org/papers/volume15/delgado14a/delgado14a.pdf) compared 179 classifiers on 121 datasets:
  - First best class of methods was Random Forest and second best class of methods was (RBF) SVMs.
- If you like to read original papers here (https://www.stat.berkeley.edu/~breiman/randomforest2001.pdf) is the original paper on Random Forests by Leo Breiman.

True or False questions on Random Forests (Class discussion)

1. Every tree in a random forest uses a different bootstrap sample of the training set.
2. To train a tree in a random forest, we first randomly select a subset of features. The tree is then restricted to only using those features.
3. A reasonable implementation of `predict_proba` for random forests would be for each tree to "vote" and then normalize these vote counts into probabilities.
4. Increasing the hyperparameter max_features (the number of features to consider for a split) makes the model more complex and moves the fundamental tradeoff toward lower training error.
5. A random forest with only one tree is likely to get a higher training error than a decision tree of the same depth.

How would you carry out "soft voting" with `predict_proba` output instead of hard voting for random forests?